

# Python: module cdms.MV

## *cdms.MV*

[index](#)

CDMS Variable objects, MaskedArray interface

### *Modules*

[MA](#)

[Numeric](#)

[string](#)

### *Classes*

[var\\_binary\\_operation](#)  
[var\\_unary\\_operation](#)

class *var\_binary\_operation*

Methods defined here:

*\_\_call\_\_*(self, a, b)

*\_\_init\_\_*(self, mafunc)

[var\\_binary\\_operation](#)(mafunc)

mafunc is an MA masked\_binary\_function.

*accumulate*(self, target, axis=0)

*outer*(self, a, b)

Return the function applied to the outer product of a and b

*reduce*(self, target, axis=0)

class *var\_unary\_operation*

Methods defined here:

*\_\_call\_\_*(self, a)

*\_\_init\_\_*(self, mafunc)

[var\\_unary\\_operation](#)(mafunc)

mafunc is an MA masked\_unary\_function.

## Functions

**`arange`** = arrayrange(start, stop=None, step=1, typecode=None, axis=None, attributes=None, id=None)  
Just like range() except it returns a variable whose type can be specified by the keyword argument typecode. The axis of the result variable

**`argsort`**(x, axis=-1, fill\_value=None)  
Treating masked values as if they have the value fill\_value, return sort indices for sorting along given axis.  
if fill\_value is None, use fill\_value(x)

**`arrayrange`**(start, stop=None, step=1, typecode=None, axis=None, attributes=None, id=None)  
Just like range() except it returns a variable whose type can be specified by the keyword argument typecode. The axis of the result variable

**`asarray`**(data, typecode=None)  
`asarray`(data, typecode=None) = array(data, typecode=None, copy=0)  
Returns data if typecode if data is a MaskedArray and typecode None or the same.

**`average`**(a, axis=0, weights=None, returned=0)  
`average`(a, axis=0, weights=None)  
Computes average along indicated axis.  
If axis is None, average over the entire array  
Inputs can be integer or floating types; result is of type Float.  
  
If weights are given, result is `sum(a*weights) / (sum(weights)*1.0)`  
weights must have a's shape or be the 1-d with length the size of a in the given axis.  
  
If returned, return a tuple: the result and the sum of the weights or count of values. Results will have the same shape.  
  
masked values in the weights will be set to 0.0

**`choose`**(indices, t)  
Returns an array shaped like indices containing elements chosen from t.  
If an element of t is the special element masked, any element of the result that "chooses" that element is masked.  
  
The result has only the default axes.

**`commonAxes`**(a, bdom, omit=None)  
Helper function for commonDomain. 'a' is a variable or array, 'b' is an axislist or None.

**`commonDomain`**(a, b, omit=None)  
`commonDomain`(a,b) tests that the domains of variables/arrays a and b and returns the common domain if equal, or None if not equal. The two differ in that one domain may have leading axes not common

to the other; the result domain will contain those axes.  
If <omit> is specified, as an integer i, skip comparison of the ith dimension and return None for the ith (common) dimension.

#### ***commonGrid(a, b, axes)***

commonGrid(a,b,axes) tests if the grids associated with variables and consistent with the list of axes. If so, the common grid is returned. a and b can be Numeric arrays, in which case the result is returned.

The common grid is 'consistent' with axes if the grid axes (e.g., longitude coordinate variables) are members of the list 'axes'.

If the grid(s) of a, b are rectilinear, the result is None, as they are defined by the axes.

#### ***commonGrid1(a, gb, axes)***

Helper function for commonGrid.

#### ***concatenate(arrays, axis=0, axisid=None, axisattributes=None)***

Concatenate the arrays along the given axis. Give the extended axis attributes provided - by default, those of the first array.

#### ***count(a, axis=None)***

Count of the non-masked elements in a, or along a certain axis.

#### ***diagonal(a, offset=0, axis1=0, axis2=1)***

diagonal(a, offset=0, axis1=0, axis2 = 1) returns the given diagonals defined by the two dimensions of the array.

#### ***fromfunction(f, dimensions)***

Apply f to s to create an array as in Numeric.

#### ***fromstring(s, t)***

Construct a masked array from a string. Result will have no mask. t is a typecode.

#### ***isMaskedVariable(x)***

Is x a masked variable, that is, an instance of AbstractVariable?

#### ***left\_shift(a, n)***

Left shift n bits

#### ***masked\_array(a, mask=None, fill\_value=None, axes=None, attributes=None, id=None)***

masked\_array(a, mask=None) =  
array(a, mask=mask, copy=0, fill\_value=fill\_value)  
Use fill\_value(a) if None.

#### ***masked\_equal(x, value)***

masked\_equal(x, value) = x masked where x == value  
For floating point consider masked\_values(x, value) instead.

#### ***masked\_greater(x, value)***

```

masked_greater(x, value) = x masked where x > value

masked_greater_equal(x, value)
masked greater equal(x, value) = x masked where x >= value

masked_inside(x, v1, v2)
x with mask of all values of x that are inside [v1,v2]

masked_less(x, value)
masked less(x, value) = x masked where x < value

masked_less_equal(x, value)
masked less equal(x, value) = x masked where x <= value

masked_not_equal(x, value)
masked not equal(x, value) = x masked where x != value

masked_object(data, value, copy=1, savespace=0, axes=None, attributes=None, id=None)
Create array masked where exactly data equal to value

masked_outside(x, v1, v2)
x with mask of all values of x that are outside [v1,v2]

masked_values(data, value, rtol=1.0000000000000001e-05, atol=1e-08, copy=1, savespace=0, axes=None)
masked values(data, value, rtol=1.e-5, atol=1.e-8)
Create a masked array; mask is None if possible.
May share data values with original array, but not recommended.
Masked where abs(data-value) <= atol + rtol * abs(value)

masked_where(condition, x, copy=1)
Return x as an array masked where condition is true.
Also masked where x or condition masked.

ones(shape, typecode='l', savespace=0, axes=None, attributes=None, id=None, grid=None)
ones(n, typecode=Int, savespace=0, axes=None, attributes=None, id=None)
an array of all ones of the given length or shape.

outerproduct(a, b)
outerproduct(a,b) = {a[i]*b[j]}, has shape (len(a), len(b))

power(a, b, third=None)
a**b

product(a, axis=0)
Product of elements along axis.

repeat(a, repeats, axis=0)
repeat elements of a repeats times along axis
repeats is a sequence of length a.shape[axis]
telling how many times to repeat each element.

reshape(a, newshape, axes=None, attributes=None, id=None, grid=None)

```

Copy of a with a new shape.

**resize(a, new\_shape, axes=None, attributes=None, id=None, grid=None)**  
resize(a, new\_shape) returns a new array with the specified shape.  
The original array's total size can be any size.

**right\_shift(a, n)**  
Right shift n bits

**set\_default\_fill\_value(value\_type, value)**  
Set the default fill value for value\_type to value.  
value\_type is a string: 'real', 'complex', 'character', 'integer', or  
value should be a scalar or single-element array.

**sort(a, axis=-1)**  
If x does not have a mask, return a masked array formed from the  
result of Numeric.sort(x, axis).  
Otherwise, fill x with fill\_value. Sort it.  
Set a mask where the result is equal to fill\_value.  
Note that this may have unintended consequences if the data contains  
fill value at a non-masked site.  
  
If fill\_value is not given the default fill value for x's type  
is used.  
The sort axis is replaced with a dummy axis.

**sum(a, axis=0, fill\_value=0)**  
Sum of elements along a certain axis.

**take(a, indices, axis=0)**  
take(a, indices, axis=0) returns selection of items from a.

**transpose(a, axes=None)**  
transpose(a, axes=None) reorder dimensions per tuple axes

**where(condition, x, y)**  
where(condition, x, y) is x where condition is true, y otherwise

**zeros(shape, typecode='l', savespace=0, axes=None, attributes=None, id=None, grid=None)**  
zeros(n, typecode=Int, savespace=0, axes=None, attributes=None, id=None)  
an array of all zeros of the given length or shape.

## Data

**Character** = 'c'  
**Complex** = 'D'  
**Complex0** = 'F'  
**Complex16** = 'F'  
**Complex32** = 'F'  
**Complex64** = 'D'  
**Complex8** = 'F'

```
Float = 'd'
Float0 = 'f'
Float16 = 'f'
Float32 = 'f'
Float64 = 'd'
Float8 = 'f'
Int = 'l'
Int0 = '1'
Int16 = 's'
Int32 = 'i'
Int8 = '1'
PyObject = 'O'
UInt = 'u'
UInt16 = 'w'
UInt32 = 'u'
UInt8 = 'b'
UnsignedInt16 = 'w'
UnsignedInt32 = 'u'
UnsignedInt8 = 'b'
UnsignedInteger = 'u'
absolute = <cdms.MV.var_unary_operation instance>
add = <cdms.MV.var_binary_operation instance>
alltrue = <cdms.MV.var_unary_operation instance>
arccos = <cdms.MV.var_unary_operation instance>
arcsin = <cdms.MV.var_unary_operation instance>
arctan = <cdms.MV.var_unary_operation instance>
arctan2 = <cdms.MV.var_binary_operation instance>
around = <cdms.MV.var_unary_operation instance>
bitwise_and = <cdms.MV.var_binary_operation instance>
bitwise_or = <cdms.MV.var_binary_operation instance>
bitwise_xor = <cdms.MV.var_binary_operation instance>
ceil = <cdms.MV.var_unary_operation instance>
conjugate = <cdms.MV.var_unary_operation instance>
cos = <cdms.MV.var_unary_operation instance>
cosh = <cdms.MV.var_unary_operation instance>
divide = <cdms.MV.var_binary_operation instance>
e = 2.7182818284590451
equal = <cdms.MV.var_binary_operation instance>
exp = <cdms.MV.var_unary_operation instance>
fabs = <cdms.MV.var_unary_operation instance>
floor = <cdms.MV.var_unary_operation instance>
fmod = <cdms.MV.var_binary_operation instance>
greater = <cdms.MV.var_binary_operation instance>
greater_equal = <cdms.MV.var_binary_operation instance>
hypot = <cdms.MV.var_binary_operation instance>
less = <cdms.MV.var_binary_operation instance>
less_equal = <cdms.MV.var_binary_operation instance>
log = <cdms.MV.var_unary_operation instance>
log10 = <cdms.MV.var_unary_operation instance>
logical_and = <cdms.MV.var_binary_operation instance>
logical_not = <cdms.MV.var_unary_operation instance>
logical_or = <cdms.MV.var_binary_operation instance>
```

```
logical_xor = <cdms.MV.var_binary_operation instance>
masked = array(data = 0, mask = 1, fill_value=[0,])
maximum = <cdms.MV._maximum_operation instance>
minimum = <cdms.MV._minimum_operation instance>
multiply = <cdms.MV.var_binary_operation instance>
negative = <cdms.MV.var_unary_operation instance>
nonzero = <cdms.MV.var_unary_operation instance>
not_equal = <cdms.MV.var_binary_operation instance>
pi = 3.1415926535897931
remainder = <cdms.MV.var_binary_operation instance>
sin = <cdms.MV.var_unary_operation instance>
sinh = <cdms.MV.var_unary_operation instance>
sometrue = <cdms.MV.var_unary_operation instance>
sqrt = <cdms.MV.var_unary_operation instance>
subtract = <cdms.MV.var_binary_operation instance>
tan = <cdms.MV.var_unary_operation instance>
tanh = <cdms.MV.var_unary_operation instance>
typecodes = {'Character': 'c', 'Complex': 'FD', 'Float': 'fd', 'Integer': '1sil', 'UnsignedInteger': 'bwu'}
```